

Formal Languages and Automata

Introduction

Ryan Stansifer

Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901

<http://www.cs.fit.edu/~ryan/>

1 February 2017

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma

Goal

- ▶ Understand the nature of computation

According to Peter J. Denning, the fundamental question underlying computer science is, “What can be (efficiently) automated?”

This is not to be interpreted as a question of what can be accomplished with today’s computing devices. This is not exclusively a question about the physical world. (Physicists, chemists, biologists study the material world.) The question is: what can be computed by any real or imagined means.

Here we ignore the question of what physical things can be accomplished by automata: swim, drive cars, explore extraterrestrial bodies, etc. These applications require one to faithfully capture the external world in a practical, non-material model. This translation is an important part of computing, but not the one we address here.

The mental challenge is dismissed by the layman. After all, cannot mankind think anything!?

The mental challenge is unpopular because it is not as visceral as robots, cell phones, and 3D printers.

Many prefer to learn what theory they need in order to accomplish some desire rather than study foundations in the hope that it proves useful later.

Curiosity is the key. (As it is behind all theoretical science.)

to tear the mask off nature and stare at the face of God

Sheldon Cooper, *The Big Bang Theory*

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma

History

Who asked first asked the question what can be automated?

Gottlob Frege (1880s): logic and arithmetic can be formalized

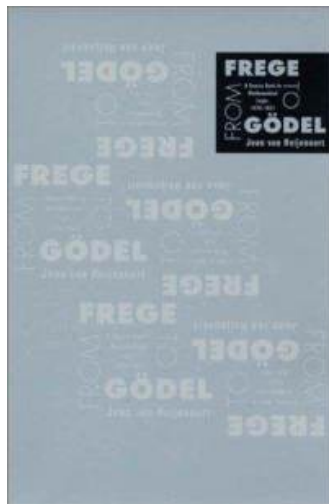
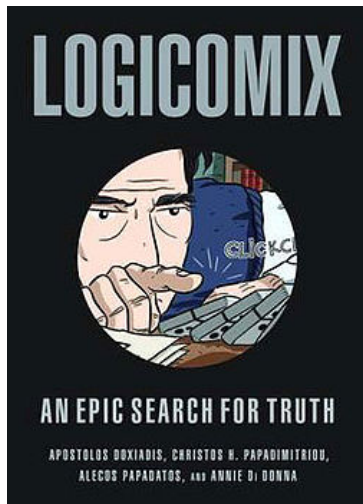
Russell (1903): not that way, but with types

Hilbert (1920s): can mathematics be formalized consistently?

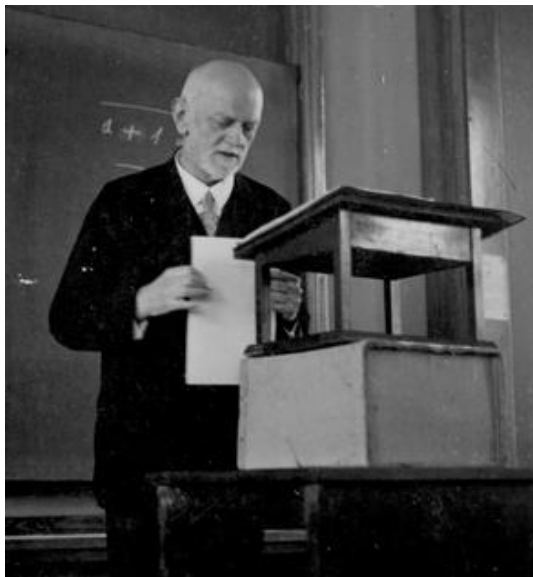
Gödel (1931): truth cannot be automated!

Then, nobody knew what could be computable. Today, we have a science of computation. So, what is computation?

Recommended Reading



David Hilbert (1863-1943)



Kurt Friedrich Gödel (1906–1978)



Kurt Friedrich Gödel (1906–1978)

Kurt Gödel's achievement in modern logic is singular and monumental - indeed it is more than a monument, it is a landmark which will remain visible far in space and time. . . . The subject of logic has certainly completely changed its nature and possibilities with Gödel's achievement.

John von Neumann

Kurt Friedrich Gödel (1906–1978)

In 1931 and while still in Vienna, Gödel published his incompleteness theorems in *Über formal unentscheidbare Sätze der "Principia Mathematica" und verwandter Systeme* (called in English "On Formally Undecidable Propositions of "Principia Mathematica" and Related Systems"). In that article, he proved for any *computable* axiomatic system that is powerful enough to describe the arithmetic of the natural numbers that:

1. If the system is consistent, it cannot be complete.
2. The consistency of the axioms cannot be proven within the system.

These theorems ended a half-century of attempts, beginning with the work of Frege and culminating in Russell and Whitehead's *Principia Mathematica* and Hilbert's formalism, to find a set of axioms sufficient for all mathematics.

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma



What is an automaton?

An automaton is

What is an automaton?

An automaton is a self-moving, self-operating machine.

In our (scientific, mathematical) context we are less interested in machines which can be realized mechanically, and more interested in abstract or virtual machines.

But first, some history of automata.

Antikythera Mechanism – 200 BC



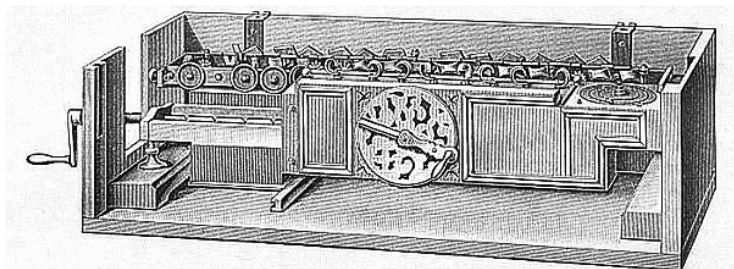
The Antikythera Mechanism

Clockwork – Installed 1410

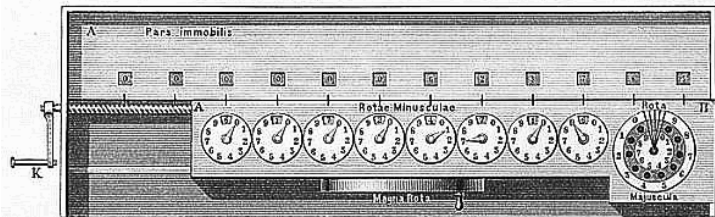


Watch the Astronomical Clock Prague on YouTube.

Stepped Reckoner – 1670s



2. Rechenmaschine von Leibniz (1673, Hannover).



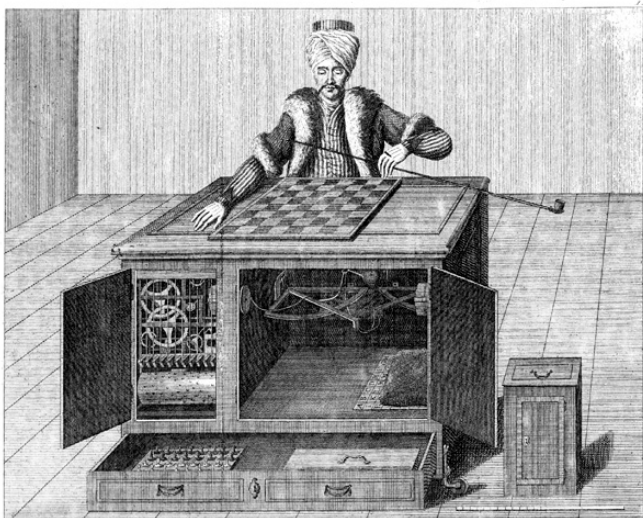
3. Leibnizsche Rechenmaschine, geometrische Zeichnung.

Leibniz Stepped Reckoner

Also, Search for Truth Calculus

Gottfried Wilhelm Leibniz (1646–1716) searched for a method he called *characteristica generalis* or *lingua generalis*.

I would like to give a method . . . in which all truths of the reason would be reduced to a kind of calculus. This could at the same time be a kind of language or universal script, but very different from all that have been projected hitherto, because the characters and even the words would guide reason, and the errors (except those of fact) would only be errors of computation. It would be very difficult to form or invent this Language or Characteristic, but very easy to learn it without any Dictionaries.



W. de Kempelen del.

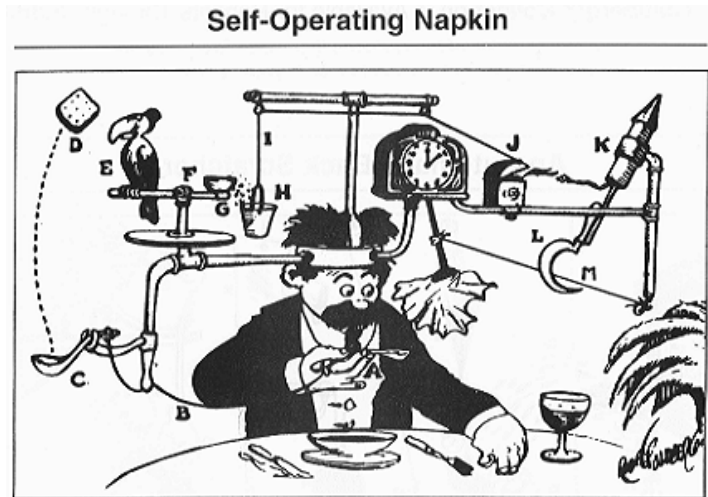
Ch. a. Mehel. excud. Basilea.

P. G. Piaty. fecit.

Der Schach-Spieler, wie er vor dem Spiel, die Figuren wie er vor dem Spiel, den Schach, tel qu'on le montre avant le jeu, par devant.

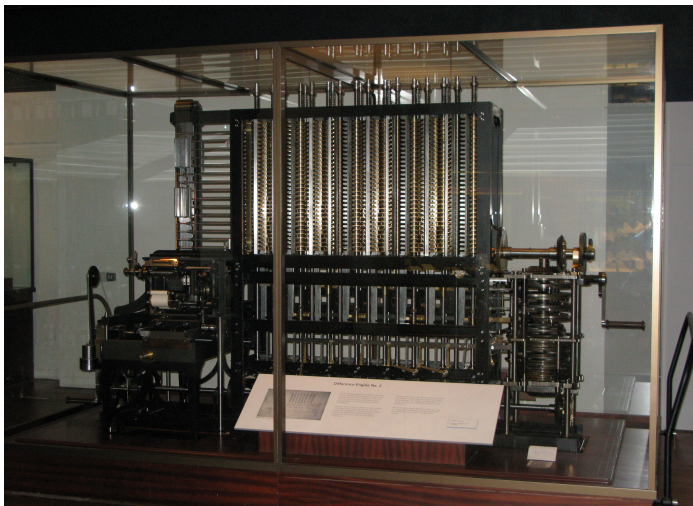
Mechanical Turk (ca1775–1850) from the BBC

Rube Goldberg (US cartoonist 1883–1970)



Sesame Street on YouTube

Difference Engine – 1820s



The difference engine by Charles Babbage

Vending Machine



End of the historical examples.

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma



What is an *abstract* machine?

A possible literary example. *Das Glasperlenspiel* (*The Glass Bead Game*). Hermann Hesse did not describe precise rules, and, indeed, probably could not comprehend a completely formal game.

Humanity requires ambiguity, contradiction, etc.

Obviously, complex problems have computer programs that solve them. But to most people this is magic and provides no evidence of a science of computation.

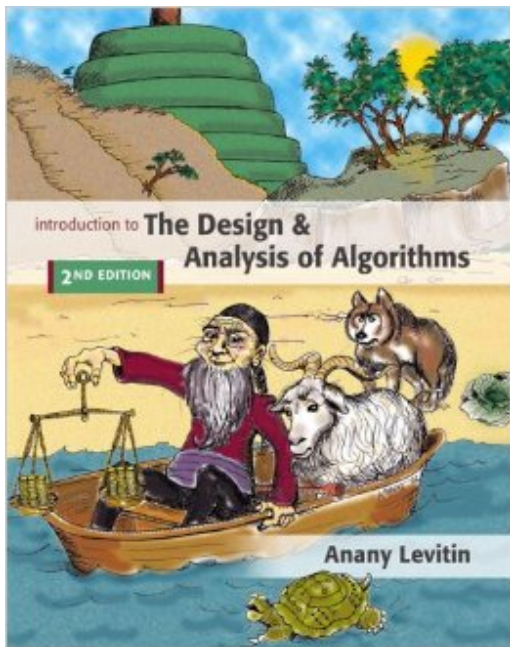
“Any sufficiently advanced technology is indistinguishable from magic.”

Arthur C. Clarke

The science of computing can be best be appreciated by building up the little pieces bottom-up.

So, identify simple, indisputable pieces and see how far you can go.

Take a simple example of problem solving: wolf, goat, cabbage.



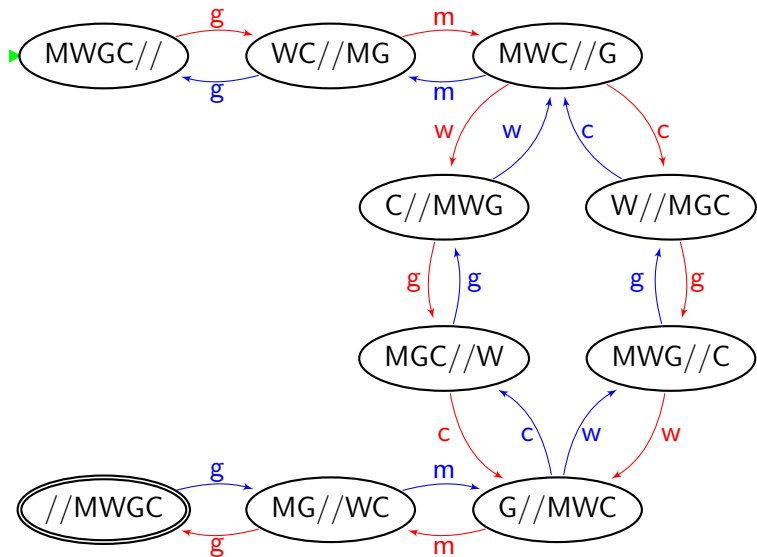
The cabbage, goat, wolf problem has simple actions.

- c row cabbage to the other side
- g row goat to the other side
- w row wolf to the other side
- m row alone to the other side

And, it has two constraints. When left unattended, the wolf will eat the goat, and the goat will eat the cabbage.

How, then, can the cabbage, the goat, and the wolf get to the other side?

A state, transition diagram can help.

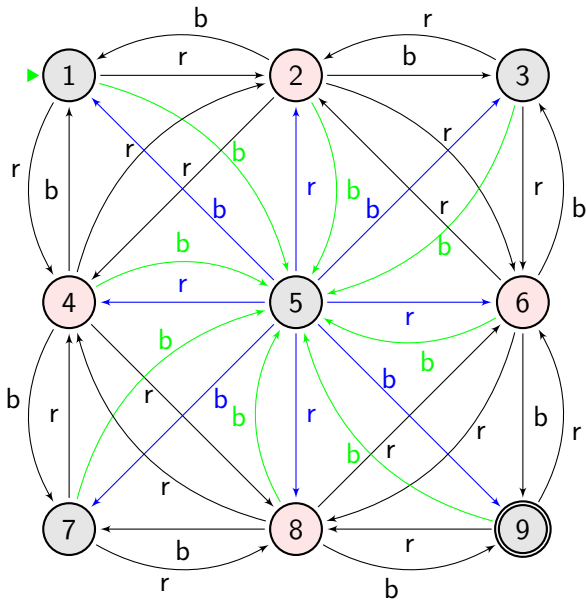


Model of a Simple Game

1	2	3
4	5	6
7	8	9

A checkerboard (rectangular lattice with 8 neighbors) with (non-deterministic) moves to the adjacent red squares or black squares.

- r move to some adjacent red square
- b move to some adjacent black square



Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma

Deterministic Finite Automata

Although we will take up the definition in more detail later, we present the formal definition of the first automata we shall study.

Deterministic Finite Automata

HMU, 3rd. Section 2.2.1 Definition of a Deterministic Finite Automaton, page 45

HMU, 3rd. Section 2.3.2 Definition of Nondeterministic Finite Automata, page 57

HMU, 3rd. Section 2.5.2 The Formal Notation of an ϵ -NFA, page 73

Linz, 6th. Definition 2.1 Deterministic finite acceptor, page 39

Linz, 6th. Definition 2.4 Nondeterministic finite acceptor, page 51

Deterministic Finite Automata

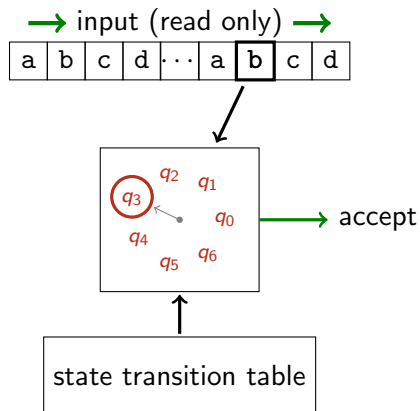
A deterministic finite automata is:

$$\langle Q, \Sigma, \delta, q_0, F \rangle$$

- ▶ Q is a finite set of states
- ▶ Σ is a finite alphabet (set of symbols)
- ▶ $\delta : Q \times \Sigma \rightarrow Q$ is a transition function
- ▶ $q_0 \in Q$ is a distinguished start state
- ▶ $F \subseteq Q$ is a set of final states

The general purpose of automata is to represent computations—compute answers. First, we examine the pieces of the formal definition, then we explain how it computes.

A Finite Automaton



States

Q is a finite set of symbols denoting abstract states.

An individual state could represent:

- ▶ The man and the cabbage are on the west bank of the river.
- ▶ It is noon EST.
- ▶ Twenty cents have been deposited in the vending machine.
- ▶ The first die shows 2 pips, and the second die shows 5 pips.
- ▶ All five dice show the same number of pips.
- ▶ The content of the computer's registers is ...
- ▶ The overflow flag is set.

The symbols, names symbols of the state, are the only thing that matters to the abstract machine, not what the state *signifies*.

Some states are distinguished by being initial or final states.

Alphabet

Σ is a finite set of symbols called the alphabet. The problem input must be encoded in the alphabet.

In our digital world we are accustomed to everything from books to movies being encoded as zeros and ones.

In fact $\Sigma = \{0, 1\}$ and $\Sigma = \{a, b\}$ are often used in our examples, because they are simple alphabets. More complicated alphabets (Latin-0 or Unicode) do not allow us to express more in theory. Though they may be more convenient in practice.

The choice of alphabet does not have any impact on our theory.

Output?

Our machine has input: some string of symbols from the alphabet. We will use our automata as accepters, namely, execution of the machine will be a simple “yes” or “no.”

This appears to be a severe limitation on computation. We can compute everything using yes and no questions.

- ▶ Is $1 + 2 = 3$? “Yes, it is.”
- ▶ Is $1 + 2 = 4$? “No, it isn’t.”

$f(x) = y$ if, and only if $xR_f y$

Transitioning

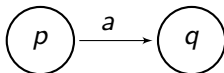
The crux of the machine is its operation which is precisely described by the transition function commonly denoted by the Greek letter δ . The domain of the transition function δ is $Q \times \Sigma$ and the range is Q . (NB. A function is just a special case of a relation, and so we might well allow an arbitrary relation as indeed we do later.) The range of δ is finite and so there only a finite number of values the function defines.

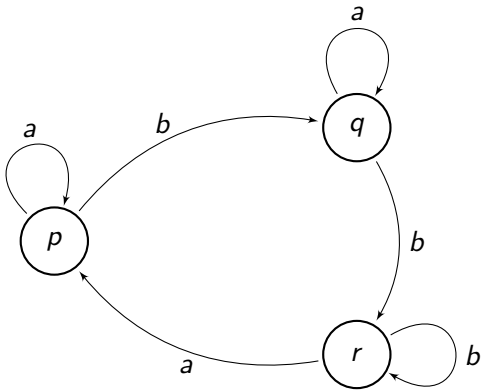
A labeled, graph may be the best way to communicate the transition function of a finite automata to a person, though sometimes these graphs can be convoluted.

The transition function may also be communicate by means a table. Different kinds of tables can be used to express the same transition function.

A Single Transition

We sometimes write a single transition from state p to state q on input symbol a in this manner: $p \xrightarrow{a} q$. Often the states are depicted as nodes in a graph, as in the following:





Transition Tables

current state	input char	next state
<i>p</i>	<i>a</i>	<i>p</i>
<i>p</i>	<i>b</i>	<i>q</i>
<i>q</i>	<i>a</i>	<i>q</i>
<i>q</i>	<i>b</i>	<i>r</i>
<i>r</i>	<i>a</i>	<i>p</i>
<i>r</i>	<i>b</i>	<i>r</i>

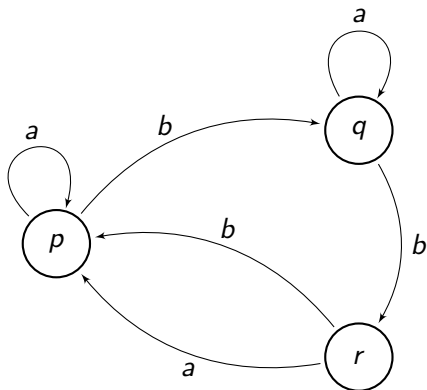
Transition Tables

	<i>a</i>	<i>b</i>
<i>p</i>	<i>p</i>	<i>q</i>
<i>q</i>	<i>q</i>	<i>r</i>
<i>r</i>	<i>p</i>	<i>r</i>

	<i>p</i>	<i>q</i>	<i>r</i>
<i>p</i>	<i>a</i>	<i>b</i>	
<i>q</i>		<i>a</i>	<i>b</i>
<i>r</i>	<i>a</i>		<i>b</i>

	<i>p</i>	<i>q</i>	<i>r</i>
<i>p</i>	<i>a</i>	<i>b</i>	\emptyset
<i>q</i>	\emptyset	<i>a</i>	<i>b</i>
<i>r</i>	<i>a</i>	\emptyset	<i>b</i>

Multi-edge Transition Graph



Transition Functions

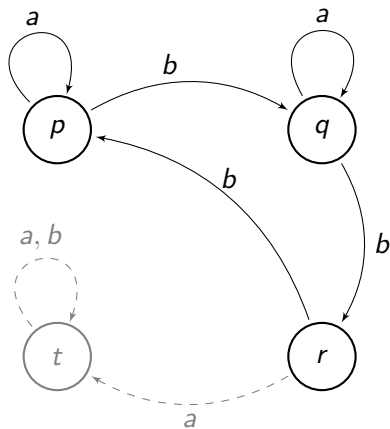
current state	input char	next state
p	a	p
p	b	q
q	a	q
q	b	r
r	a	p
r	b	p

	a	b
p	p	q
q	q	r
r	p	p

	p	q	r
p	a	b	
q		a	b
r	a, b		

	p	q	r
p	a	b	\emptyset
q	\emptyset	a	b
r	$a + b$	\emptyset	\emptyset

Transition Graph With Stuck State



Transition Functions

current state	input char	next state
p	a	p
p	b	q
q	a	q
q	b	r
r	a	t
r	b	p
t	a	t
t	b	t

	a	b		p	q	r	t
p	p	q	p	a	b		
q	q	r	q		a	b	
r	t	p	r	b			a
t	t	t	t				a, b

	p	q	r	t
p	a	b	\emptyset	\emptyset
q	\emptyset	a	b	\emptyset
r	b	\emptyset	\emptyset	a
t	\emptyset	\emptyset	\emptyset	$a + b$

Transition Functions

A row in the table for every state and input symbol combination.

q_o	a	q_i
q_o	b	q_j
q_i	a	q_k
q_i	b	q_l

Transition Functions

A row in the table for every state and a column for every input symbol. (A missing entry in the table signals a transition to a non-final “trap” or “sink.”)

	<i>a</i>	<i>b</i>
<i>q</i> ₀	<i>q</i> _{<i>i</i>}	<i>q</i> _{<i>j</i>}
<i>q</i> ₁	<i>q</i> _{<i>k</i>}	<i>q</i> _{<i>l</i>}
<i>q</i> ₂	<i>q</i> _{<i>i</i>}	<i>q</i> _{<i>j</i>}

This form is very natural for encoding the transition function as two-dimensional array in a programming language.

Transition Functions

Another way of representing the transition function is related to the Boolean, adjacent matrix use to represent graphs. Here the matrix is square with each row and each column representing a state.

Rows are the transitions *from* the state; columns are the transitions *to* the state. The matrix entry describes the input symbol

	q_0	q_i	q_j	q_k	q_l
q_0	a		b	a	a
q_i	a		b	a	a
q_j		b	a	a	
q_k	a		b		a
q_l	a		b	a	

Blank cells mean that there is no transition on any input symbol between the two state.

But what about a transition from a state to another state on more than one symbol of the alphabet? (This is related to the problem caused by multi-graphs—those with parallel edges.)

Transition Functions

Transitions on multiple

	q_0	q_i	q_j	q_k	q_l
q_0	a		b	a	a
q_i	a		$a + b$	a	a
q_j		b	a	a	
q_k	a		b		$a + b$
q_l	a		b	a	

Notice the similarity to regular expressions!

Transition Functions

	q_0	q_i	q_j	q_k	q_l
q_0	a	\emptyset	b	a	a
q_i	a	\emptyset	$a + b$	a	a
q_j	\emptyset	b	a	a	\emptyset
q_k	a	\emptyset	b	\emptyset	$a + b$
q_l	a	\emptyset	b	a	\emptyset

Consider the possibility of marking the cells of the matrix (the transitions of the automaton) with arbitrary regular expressions.

Encoding Finite Automata

We can encode an automaton as a transition matrix; a two-dimensional array indexed by state number and input character. There will be a “dead” state (state 0) that loops to itself on all characters; we use this state to encode the absence of an edge.

```
int edges [] [] = { /* ... 0 1 2 ... e f g h i ... */
/* state 0 */ {0,0,... 0,0,0,... 0,0,0,0,0,...},
/* state 1 */ {0,0,... 7,7,7,... 4,4,4,4,2,...},
/* state 2 */ {0,0,... 4,4,4,... 4,3,4,4,4,...},
/* state 3 */ {0,0,... 4,4,4,... 4,4,4,4,4,...},
/* state 4 */ {0,0,... 4,4,4,... 4,4,4,4,4,...},
/* state 5 */ {0,0,... 6,6,6,... 0,0,0,0,0,...},
/* state 6 */ {0,0,... 6,6,6,... 0,0,0,0,0,...},
/* state 8 */ {0,0,... 8,8,8,... 0,0,0,0,0,...},
// and so on
}
```

We must also know which of the state is the start state and which are the final states.

Is is convenient to use row zero as the “dead” or “trap” state. It is a row of all zeroes

Code for Table-Driven Automata

```
Current_State := The_Initial_State;
while not (End_Of (Input_Stream)) loop
    Input_Char := Next_Character (Input_Stream);
    Current_State := Edges [Current_State] [Input_Char];
end loop;
if (Final_State (Current_State)) then
    Accept;
else
    Reject;
end if;
```

Can you formalize the cabbage, good, wolf problem?

$$\langle Q, \Sigma, \delta, q_0, F \rangle$$

- ▶ $Q = \{ MWGC//, \dots, //MWGC \}$
- ▶ $\Sigma = \{ c, g, w, m \}$
- ▶ $\delta : Q \times \Sigma \rightarrow Q$ is a transition function
- ▶ $q_0 = MWGC//$
- ▶ $F = \{ //MWGC \}$

	c	g	w	m
0	0	0	0	0
$MWGC//$	0	$WC//MG$	0	0
$WC//MG0$	$MWGC//$	0	$MWC//G$	

We have examined the “hardware” of the DFA, but we have not said anything about how it is used.

It is used to define a set of strings. This seems ridiculously simple and abstract.

It is abstract. In this way the computation has been distilled to its essential nature. Information is encoded in strings and computation into yes and no questions.

So, the question becomes how does an automaton define a formal language.

Extended Transition Function

Let M be the deterministic finite automaton $\langle Q, \Sigma, \delta, q_0, F \rangle$.

Define extended transition function $\delta^* : Q \times \Sigma^* \rightarrow Q$ for M by:

$$\begin{aligned}\delta^*(q, \epsilon) &= q \\ \delta^*(q, a : w) &= \delta^*(\delta(q, a), w)\end{aligned}$$

The symbol a is the next symbol to be read (the symbol under the “read head” of the machine); the string w is the future string to be read.

This definition works fine if δ is a partial function on $Q \times \Sigma$, then δ^* is partial as well.

The formal language defined by the machine M is denoted $L(M)$ and is defined as follows:

$$L(M) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$$

Deterministic Finite Automata

An equivalent approach is more general

Deterministic Finite Automata

Let M be the deterministic finite automaton $\langle Q, \Sigma, \delta, q_0, F \rangle$. We define an *instantaneous description* or ID of M to be the pair $\langle q, w \rangle$ where $q \in Q$ is a state and $w \in \Sigma^*$ is a string representing the unread input.

We define a binary relation \vdash , called the transition relation, on the set of IDs

$$\langle q, aw \rangle \vdash \langle q', w \rangle \quad \text{if } \delta(q, a) = q'$$

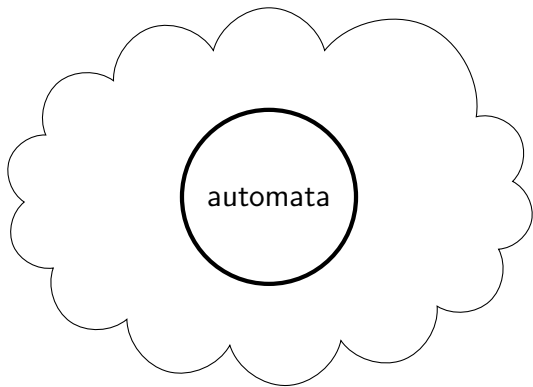
The binary relation \vdash^* , called the reachability relation, is the reflexive, transitive closure of transition relation, \vdash .

$$L(M) = \{w \in \Sigma^* \mid \langle q_0, w \rangle \vdash^* \langle q_f, \epsilon \rangle \text{ with } q_f \in F\}$$

Deterministic Finite Automata

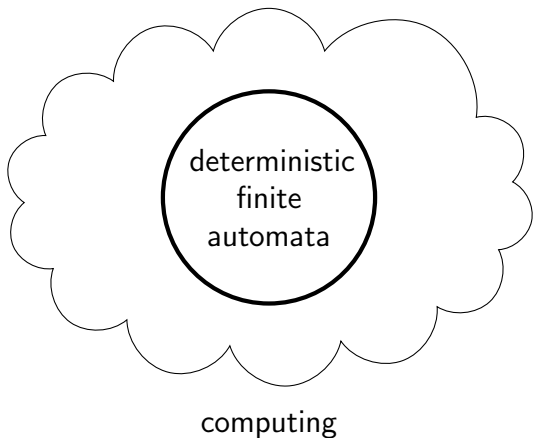
The inductive definition of the reachability relation means that induction can be used to prove properties about IDs and hence on the set of strings recognized by an automata.

$$\frac{}{\langle q, w \rangle \vdash^* \langle q, w \rangle} \quad \frac{\langle q, w \rangle \vdash \langle q', w' \rangle \quad \langle q', w' \rangle \vdash^* \langle q'', w'' \rangle}{\langle q, w \rangle \vdash^* \langle q'', w'' \rangle}$$



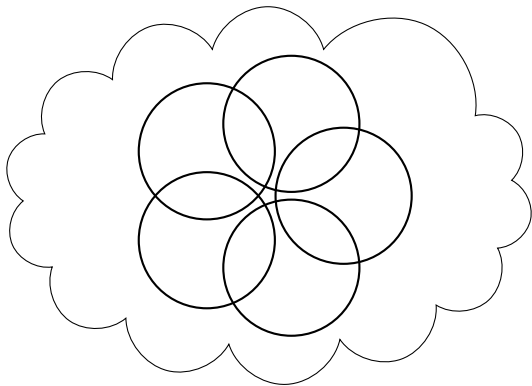
computing

Whatever an automaton is, it should certainly be a simple model of computation without any doubts.

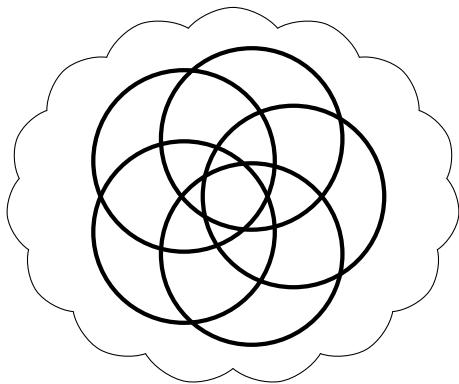


Being simple, deterministic finite automata certainly are a possible model of computation. However, it seems quite unlikely that this encompasses all computation.

Other models, all obviously computable, can be proposed.
The result might be chaotic.

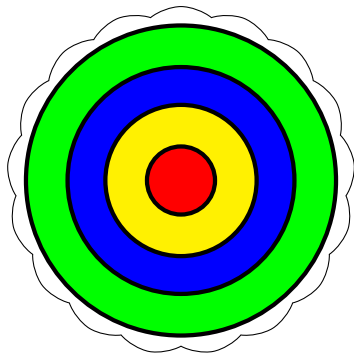


Models of computation might have little relationship to each other.



In fact, a clear picture (science) emerges.

Something like the following picture.



Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma

What if we consider two-way, deterministic finite automata (2DFA)? (Two-way automata can re-read the input.) Turns out that they are the same as one-way deterministic finite automata. (We must carefully formalize what it means for two machines to solve the same class of problems.) More interesting are other more radical variations, like non-deterministic finite automata.

Nondeterministic Finite Automata

A nondeterministic finite automata is:

$$\langle Q, \Sigma, \Delta, q_0, F \rangle$$

- ▶ Q is a finite set of states
- ▶ Σ is a finite alphabet (set of symbols)
- ▶ $\Delta : Q \times \Sigma_\epsilon \times Q$ is a transition relation. Define $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ to be the transition function $\delta(q, \sigma) = \{q' \in Q \mid \langle q, \sigma, q' \rangle \in \Delta\}$.
- ▶ $q_0 \in Q$ is the distinguished initial state of the control unit, and
- ▶ $F \subseteq Q$ is a set of final states

we define Σ_ϵ to be $\Sigma \cup \{\epsilon\}$

Non-Deterministic Finite Automata

Let M be the non-deterministic finite automaton $\langle Q, \Sigma, \Delta, q_0, F \rangle$. We define an *instantaneous description* or ID of M to be the pair $\langle q, w \rangle$ where $q \in Q$ is a state and $w \in \Sigma^*$ is a string representing the unread input.

For $q \in Q$ and $a \in \Sigma_\epsilon$ we define a binary relation \vdash on the set of IDs

$$\langle q, aw \rangle \vdash \langle q', w \rangle \quad \text{if } \langle q, a, q' \rangle \in \Delta$$

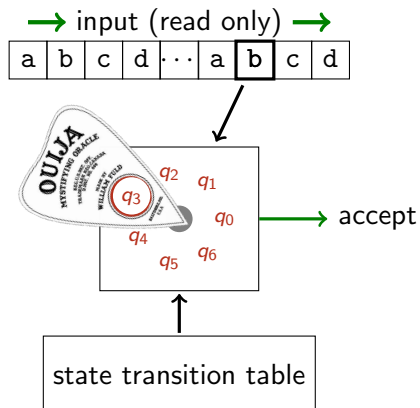
This includes as a special case:

$$\langle q, w \rangle \vdash \langle q', w \rangle \quad \text{if } \langle q, \epsilon, q' \rangle \in \Delta$$

The binary relation \vdash^* is the reflexive, transitive closure of \vdash . So, now we let

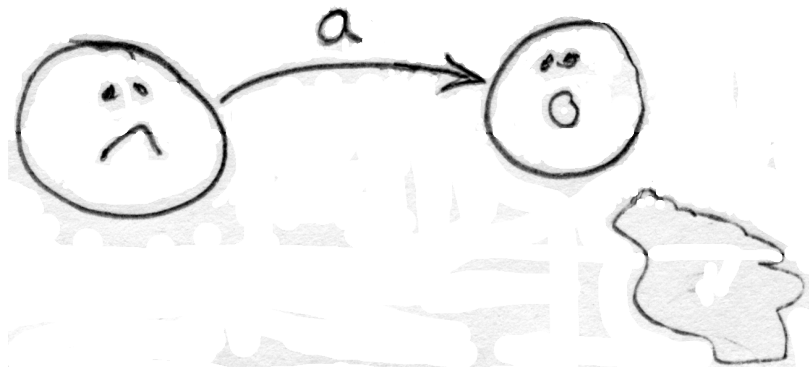
$$L(M) = \{w \in \Sigma^* \mid \langle q_0, w \rangle \vdash^* \langle q_f, \epsilon \rangle \text{ for any } q_f \in F\}$$

Non-Deterministic Finite Automata



Same as DFA, but needs oracle

Automata Versus Expressions



Automata Versus Expressions

$a \cdot b + c^*$

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma

Pushdown Automata

HMU, 3rd. Section 6.1.2 The Formal Definition of Pushdown Automata, page 227 HMU, 3rd, Section 6.2.1. Acceptance by Final State, page 235 HMU, 3rd, Section 6.4.1. Definition of Deterministic PDA, page 252

Linz, 6th. Definition 7.1 Nondeterministic pushdown acceptor (npda), page 183 Linz, 6th. Definition 7.2 Language accepted by a pushdown automaton, page 186 Linz, 6th. Definition 7.3 Deterministic pushdown acceptor (pda), page 203

Pushdown Automata

A pushdown automata is a 7-tuple: $\langle Q, \Sigma, \Gamma, \Delta, q_0, z, F \rangle$ where

1. Q is a finite set of states of the control unit,
2. Σ is a finite alphabet (set of symbols),
3. Γ is a finite stack alphabet (set of symbols),
4. $\Delta : Q \times \Sigma_\epsilon \times \Gamma \times Q \times \Gamma^*$ is a finite transition relation,
5. $q_0 \in Q$ is the distinguished initial state,
6. $z \in \Gamma$ is the distinguished stack start symbol, and
7. $F \subseteq Q$ is a set of final states

Let M be the nondeterministic pushdown automaton $\langle Q, \Sigma, \Gamma, \Delta, q_0, Z, F \rangle$. We define an *instantaneous description* or ID of M to be the triple $\langle q, w, \gamma \rangle$ where $q \in Q$ is the current state, $w \in \Sigma^*$ is a string representing the unread input, and γ is the stack.

For $q \in Q$, $w \in \Sigma^*$, and $\gamma \in \Gamma^*$ we define a binary relation \vdash on the set of IDs

$$\langle q, aw, X\beta \rangle \vdash \langle q', w, \alpha\beta \rangle \quad \text{if } \langle q, a, X, q', \alpha \rangle \in \Delta$$

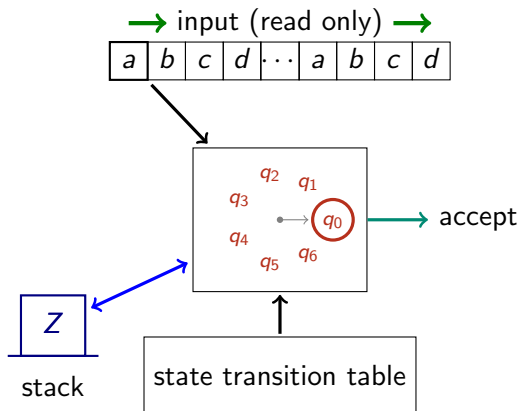
This includes as a special case:

$$\langle q, w, X \rangle \vdash \langle q', w, \alpha\beta \rangle \quad \text{if } \langle q, \epsilon, X, q', \alpha \rangle \in \Delta$$

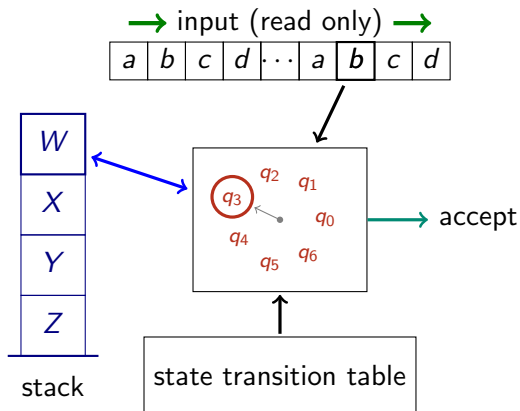
The binary relation \vdash^* is the reflexive, transitive closure of \vdash . So, now we let

$$L(M) = \{w \in \Sigma^* \mid \langle q_0, w, Z \rangle \vdash^* \langle q_f, \epsilon, \alpha \rangle \text{ for any } q_f \in F, \alpha \in \Gamma^*\}$$

Pushdown Automaton (Initial Configuration)



Pushdown Automaton



Pushdown Automata

A pushdown automata $\langle Q, \Sigma, \Gamma, \Delta, q_0, z, F \rangle$ is said to be deterministic if for all

$q \in Q, a \in \Sigma_\epsilon, \gamma \in \Gamma$ the set

$\{ \langle q, a, \gamma, q', \alpha \rangle \in \Delta \mid q' \in Q, \alpha \in \Gamma^* \}$ has cardinality one.

$\langle q_0, a, \gamma, q', \alpha \rangle$ and $\langle q_0, a, \gamma, q'', \alpha \rangle$

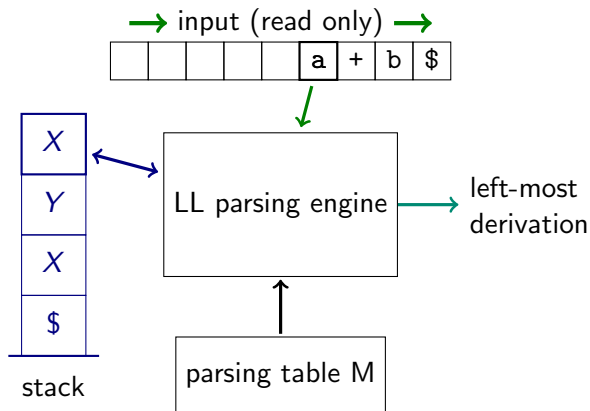
$\langle q_0, a, \gamma, q', \alpha \rangle$ and $\langle q_0, a, \gamma, q', \beta \rangle$

Hmmm.

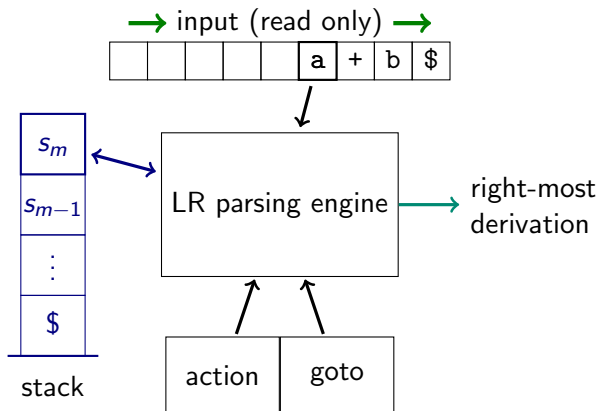
$\langle q_0, a, \gamma, q', \alpha \rangle$ and $\langle q_0, \epsilon, \gamma, q'', \alpha \rangle$

$\langle q_0, a, \gamma, q', \alpha \rangle$ and $\langle q_0, \epsilon, \gamma, q', \beta \rangle$

LL parsing: A Useful Variant



LR parsing: A Useful Variant



Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma

Theoretical computer science has gravitated around certain models of computation. For better or worse, the main one has been the Turing machine.



Alan Mathison Turing, (1912-1954) was a British pioneering computer scientist, logician, cryptanalyst, and marathon runner. He was highly influential in the development of computer science, providing a formalization of the concepts of “algorithm” and “computation” with the Turing machine, which can be considered a model of a general purpose computer. Turing is widely considered to be the father of theoretical computer science. During the Second World War, Turing worked for the Government Code and Cypher School (GC&CS) at Bletchley Park, Britain’s code-breaking center. This section played a pivotal role by enable the decryption of German messages.

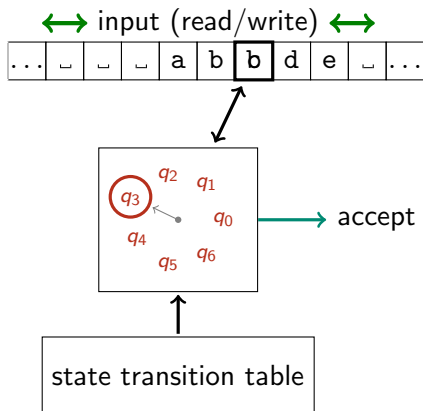
Alan Mathison Turing (1912–1954)

Turing wrote in 1936 that it is possible to invent a *single* machine which can be used to compute *any* computable sequence.

This finding is now taken for granted, but at the time it was considered astonishing. The model of computation that Turing called his “universal machine”—“U” for short—is considered by some (cf Davis (2000)) to have been the fundamental theoretical breakthrough that led to the notion of the stored program computer. In the words of Minsky (1967), page 104:

Turing’s paper ... contains, in essence, the invention of the modern computer and some of the programming techniques that accompanied it.

Turing Machine



Turing Machines

<http://www.youtube.com/watch?v=cYw2ewo06c4>

Turing Machines (Single Final State Variation)

A Turing machine is a 7-tuple $\langle Q, T, I, \delta, \sqcup, q_0, q_f \rangle$ where

1. Q is a finite set of states,
2. T is a finite set of tape symbols,
3. I is a finite set of input symbols, $I \subseteq T$,
4. $\delta : Q \times T \rightarrow Q \times T \times \{L, R\}$ is the transition function,
5. $\sqcup \in T \setminus I$ is the designated symbol for a blank (the symbol always beyond the ends of the two-way infinite tape),
6. $q_0 \in Q$ is the distinguished initial state, and
7. $q_f \in Q$ is the distinguished final or accepting state.

Turing's original paper contains a programming language, just as Gödel's paper does, or what we would now call a programming language, But these two programming languages are very different. Turing's isn't a high-level language like LISP; it's more like a machine language, the raw code of ones and zeros that are fed to a computer's central processor. Turing's invention of 1936 is, in fact, a horrible machine language, one that nobody would want to use today, because it's too rudimentary.

Gregory J. Chaitin [CHY-tən], "Computers, Paradoxes and the Foundations of Mathematics," *American Scientist*, 2002, page 168.

Furthermore

Modern stored-program computers are not accurately modeled by Turing machines. Other abstract machines such as the random access stored program machine (RASP) are closer. The RASP stores its “program” in “memory” external to its finite-state machine’s “instructions”. But unlike the Turing Machine, the RASP has an infinite number of distinguishable, numbered but unbounded “registers” or memory “cells” that can contain any integer. There are computational optimizations that can be performed based on the memory indices, which are not possible in a general Turing Machine; thus when Turing Machines are used as the basis for bounding running times, a ‘false lower bound’ can be proven on certain algorithms’ running times (due to the false simplifying assumption of a Turing Machine). An example of this is binary search, an algorithm that can be shown to perform more quickly when using the RASP model of computation rather than the Turing Machine model.

Turing Machine

Let m be the Turing machine $\langle Q, T, I, \delta, \sqcup, q_0, q_f \rangle$.

We define an *instantaneous description* or ID of M to be the triple $\langle u, q, w \rangle$ where $q \in Q$ is a state and $u, w \in T^*$ are strings. The string u is a (finite) string containing all the non-blanks symbols to the left of the read head, and the string w is a (finite) string containing all the non-blanks symbols to the right of the read head. The read head is positioned at the first character of w .

We define a binary relation \vdash on the set of IDs:

$\langle uc, q, av \rangle \vdash \langle u, q', cbv \rangle$	if $\delta(q, a) = \langle q', b, L \rangle$
$\langle uc, q, \epsilon \rangle \vdash \langle u, q', cb \rangle$	if $\delta(q, \sqcup) = \langle q', b, L \rangle$
$\langle \epsilon, q, av \rangle \vdash \langle \epsilon, q', \sqcup bv \rangle$	if $\delta(q, a) = \langle q', b, L \rangle$
$\langle \epsilon, q, \epsilon \rangle \vdash \langle \epsilon, q', \sqcup bv \rangle$	if $\delta(q, \sqcup) = \langle q', b, L \rangle$
$\langle u, q, av \rangle \vdash \langle ub, q', v \rangle$	if $\delta(q, a) = \langle q', b, R \rangle$
$\langle u, q, \epsilon \rangle \vdash \langle ub, q', \epsilon \rangle$	if $\delta(q, \sqcup) = \langle q', b, R \rangle$

The binary relation \vdash^* is the reflexive, transitive closure of \vdash .

$$L(M) = \{ w \in I^* \mid \langle q_0, w \rangle \vdash^* \langle q_f, \epsilon \rangle \}$$



Dryden Flight Research Center E49-0053 Photographed 10/49
Early "computers" at work. NASA photo



Turing modeled computation after mathematical office workers performing simple calculations on sheets of paper.

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

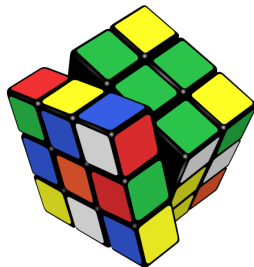
Courses

Grammars

Pumping Lemma



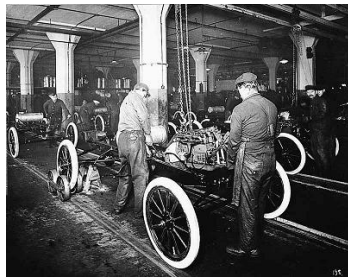
Automata compute



Expressions denote



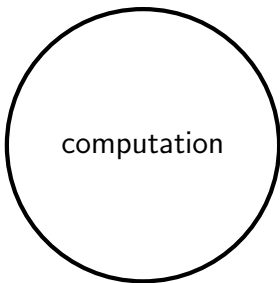
Trees demonstrate



Grammars construct

Many other computational models:

- ▶ Herbrand-Gödel μ recursive functions [5, 2],
- ▶ universal register machine,
- ▶ lambda calculus,
- ▶ combinators [16, 1]
- ▶ Post systems [13],
- ▶ and, many others.



Church-Turing Thesis:

All sufficiently powerful models of computation are equivalent!

So, we define computation as being that which can be computed by a Turing machine.

Herbrand-Gödel

Herbrand had written Gödel a letter on April 7, 1931 (see Gödel [1986, p. 368] and Sieg [1994, p. 81]), in which he wrote, “If φ denotes an unknown function, and ψ_1, \dots, ψ_k are known functions, and if the ψ 's and φ are substituted in one another in the most general fashions and certain pairs of resulting expressions are equated, then if the resulting set of functional equations has one and only one solution for φ , φ is a recursive function.” Gödel made two restrictions on this definition to make it *effective*, first that the left-hand sides of the functional equations be in standard form with φ being the outermost symbol, and second that for each set of natural numbers n_1, \dots, n_j there exists a unique m such that $\varphi(n_1, \dots, n_j) = m$ is a derived equation.

Universal Register Machines

Our mathematical idealisation of a computer is called an *unlimited register machine* (URM); it is a slight variation of a machine first conceived by Shepherdson & Sturgis [1963]. In this section we describe the URM and how it works; we begin to explore what it can do in § 3.

The URM has an infinite number of *registers* labelled R_1, R_2, R_3, \dots , each of which at any moment of time contains a natural number; we denote the number contained in R_n by r_n . This can be represented as follows

R_1	R_2	R_3	R_4	R_5	R_6	R_7	...
r_1	r_2	r_3	r_4	r_5	r_6	r_7	...

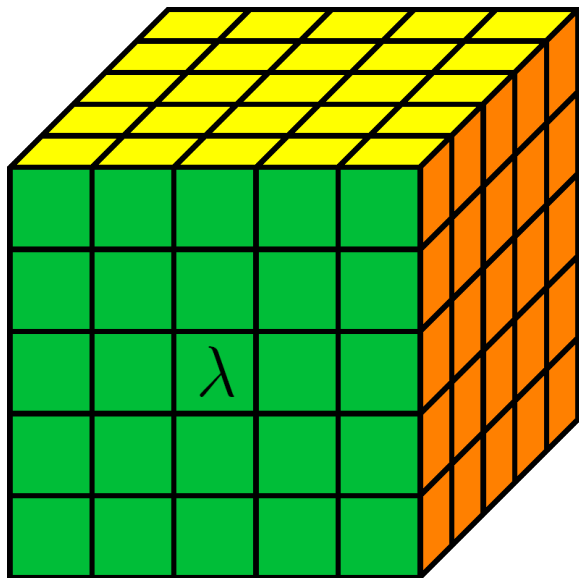
The contents of the registers may be altered by the URM in response to certain *instructions* that it can recognise. These instructions correspond to very simple operations used in performing calculations with numbers. A finite list of instructions constitutes a *program*. The instructions are of four kinds, as follows.

Lambda Calculus

Beta reduction:

$$(\lambda x. b)a \rightarrow b[x := a]$$

Lambda Calculus



Combinators

The combinatory calculus is constructed out of two combinators **S** and **K** such that

$$\mathbf{S}_{xyz} = xz(yz)$$

$$\mathbf{K}_{xy} = x$$

S is Schönfinkel's "Verschmelzungsfunktion" or "fusion" combinator and **K** is his "Konstanzfunktion" or "constancy" combinator.

However, application is still required and so this is actually a special case of the lambda calculus. Book: Hindley, J. R., and Seldin, J. P. (2008) λ -calculus and Combinators: An Introduction. Cambridge Univ. Press.

Post System

Post canonical system, or Post System.

A Post system is a quadruple $\langle \Sigma_V, \Sigma_C, P, Ax \rangle$.

Arto Salomaa, *Computation and Automata*, Cambridge University Press, 1985.

Ryan Stansifer, *The Study of Programming Languages*, Prentice-Hall, 1995.

Post Systems



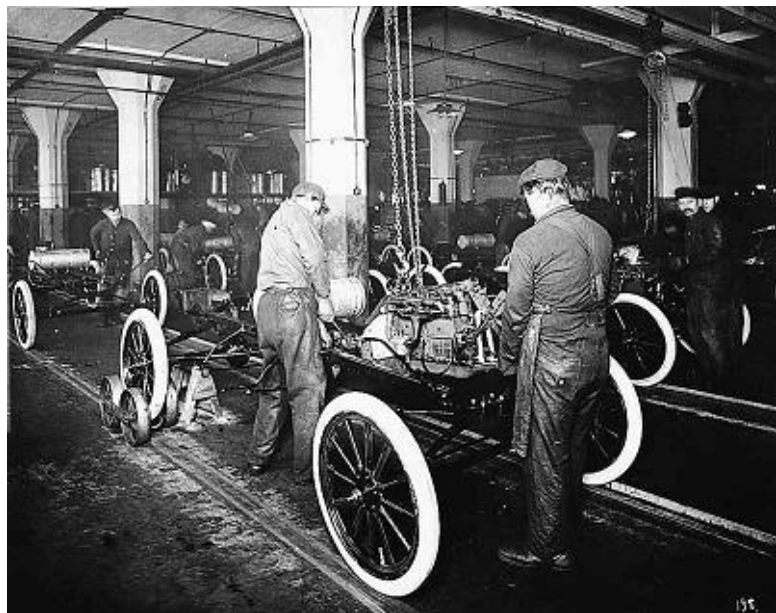
Unrestricted Grammars

A grammar is a 4-tuple $\langle T, N, P, S \rangle$:

- ▶ T is the finite set of terminal symbols;
- ▶ N is the finite set of nonterminal symbols, $T \cap N = \emptyset$, also called variables or syntactic categories;
- ▶ $S \in N$, is the start symbol;
- ▶ P is the finite set of productions.

A production has the form $\alpha \rightarrow \beta$ where α and β are strings of terminals and nonterminals (α can't be the empty string, but β might be).

Unrestricted Grammars



Many Others

Unrestricted grammars, μ -recursive functions, Markov algorithms (string rewriting system), biologically inspired models of computation (membrane systems, protein-centric interaction systems), quantum computers, and so on.

- ▶ Fernández, Maribel (2009). *Models of Computation: An Introduction to Computability Theory*. Undergraduate Topics in Computer Science. Springer. ISBN 978-1-84882-433-1.
- ▶ Savage, John E. (1998). *Models Of Computation: Exploring the Power of Computing*.

Which model is the right one?

Which model is the right one?

Church-Turing Thesis:

All sufficiently powerful models of computation are equivalent!

Which model is the right one?

Church-Turing Thesis:

All sufficiently powerful models of computation are equivalent!

How do we know? Every model proposed so far is equivalent to all the others.

Automata versus Models

I chose pictures of automata rather than their mathematical models as the pictures are more suggestive. Indeed automata (as the name suggests) are motivated by the material world as opposed to the intellectual, mathematical world. Each approach has advantages and disadvantages.

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma

Just what is in this course and why is called *Formal Languages* and Automata?

Definition. A *formal language* is a set of strings over an alphabet.

The significance is:

computational problem

=

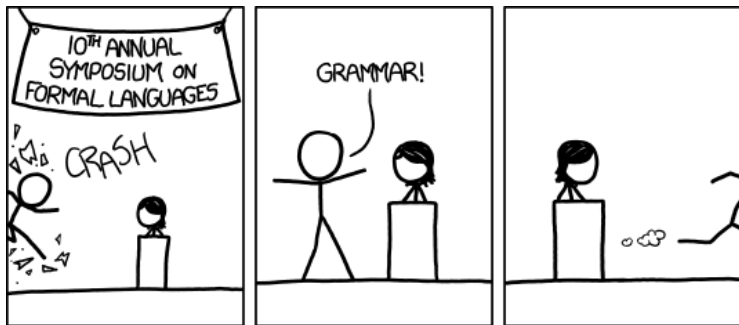
formal language

Because we can strip computation down to

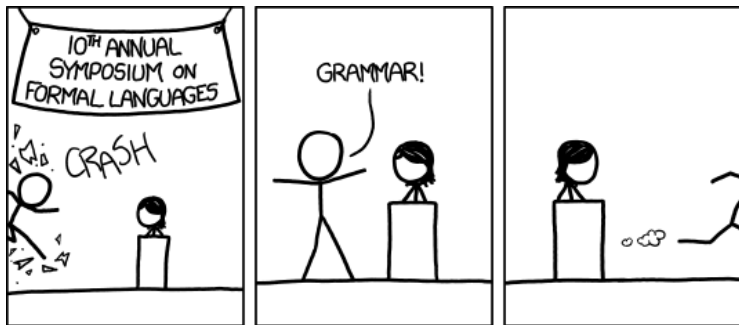
1. data – strings
2. answers – yes, or no

This approach may lack practical importance as it does not lend itself to expressing computational solutions. Neither procedural or data abstraction is convenient in this form. We take to study the essential core.

Since “problem=language”, languages and grammars get mixed up with computation.



Since “problem=language”, languages and grammars get mixed up with computation.



Title text: “[Audience looks around] ‘Just what happened?’ ‘There must have been some context we are missing.’”

See Explain XKCD 1090.

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

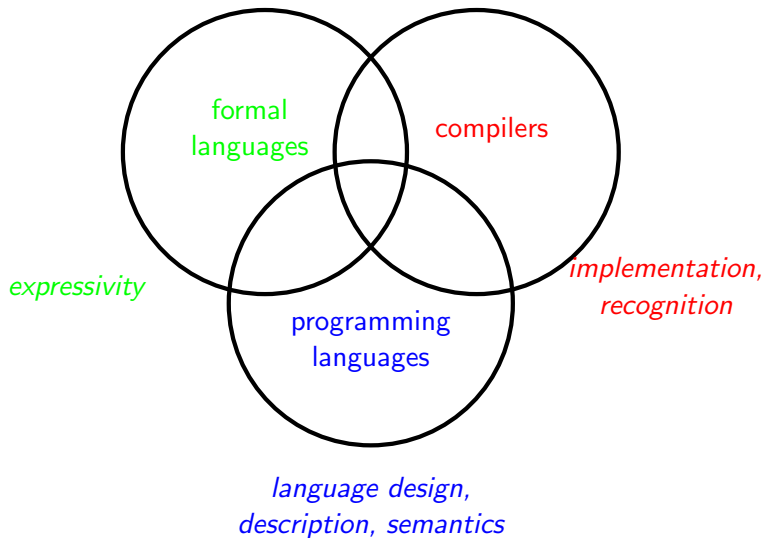
Overview

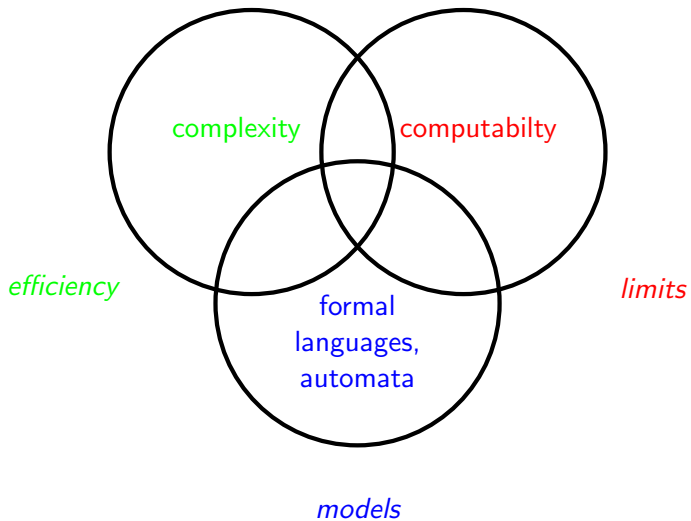
Courses

Grammars

Pumping Lemma

Different fields and different academic courses take different perspectives.



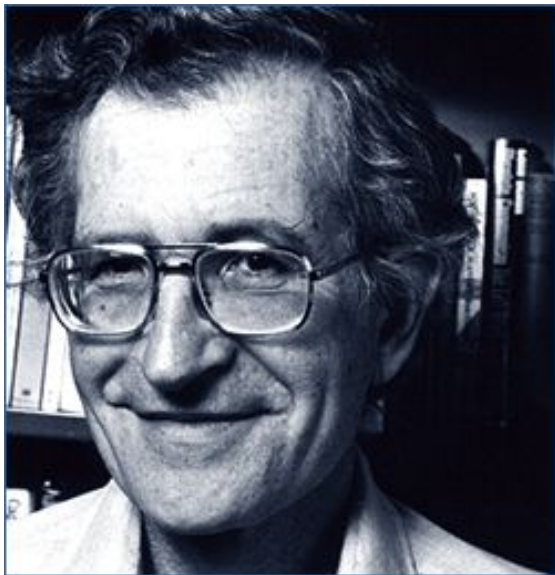


Summary

Here at the outset we summarize many of the final results.

- ▶ Chomsky hierarchy
- ▶ Recursive versus r.e.
- ▶ Closure properties of language families
- ▶ Decision algorithms

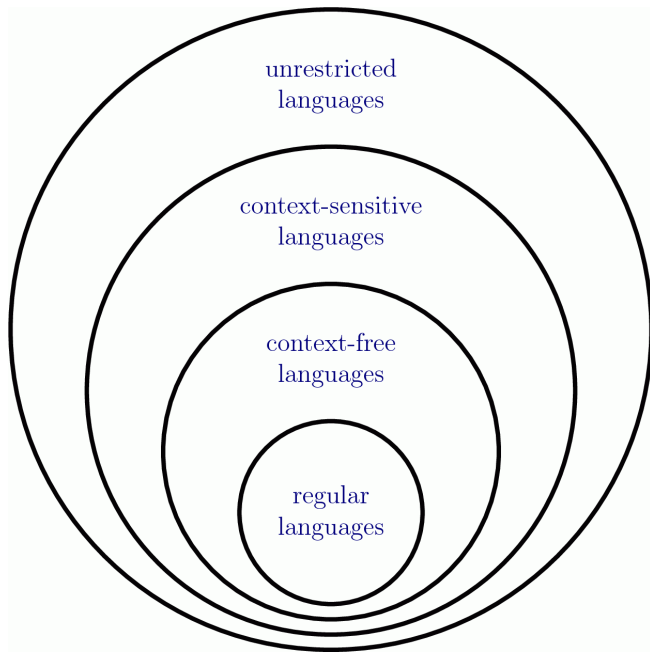
Noam Chomsky

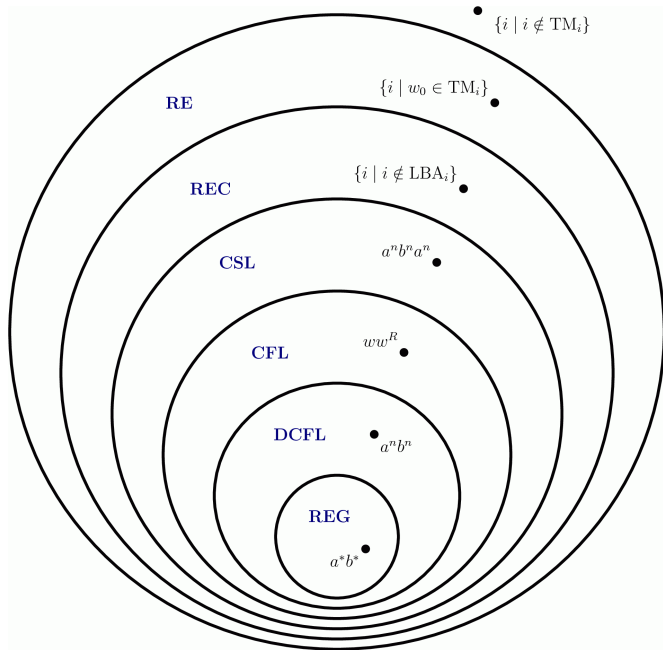


Summary

Here at the outset we summarize many of the final results.

Chomsky Hierarchy





Properties of Language Families

	REG	DCFL	CFL	CSL	REC	RE
union	✓	✗	✓	✓	✓	✓
complement	✓	✓	✗	✓	✓	✗
intersection $\overline{L_1} \cup \overline{L_2}$	✓	✗	✗	✓	✓	✓
set difference $L_1 \cap \overline{L_2}$	✓	✗	✗	✓	✓	✗
concatenation	✓	✗	✓	✓	✓	✓
Kleene star	✓	✗	✓	✓	✓	✓
intersection with REG	✓	✓	✓	✓	✓	✓

Textbooks

An introduction to some textbooks.

“Cinderella” Book

INTRODUCTION TO AUTOMATA THEORY, LANGUAGES, AND COMPUTATION

JOHN E. HOPCROFT
JEFFREY D. ULLMAN



John Hopcroft, 1986 Turing Award Recipient



INTRODUCTION TO

Automata Theory, Languages, and Computation

3rd Edition



JOHN E. HOPCROFT
RAJEEV MOTWANI
JEFFREY D. ULLMAN

An Introduction to

FORMAL LANGUAGES and AUTOMATA

Fifth Edition



PETER LINZ



INCLUDES CD-ROM



Overview of Course

The field of computer science includes a wide range of special topics, from machine design to programming. The use of computers in the real world involves a wealth of specific detail that must be learned for a successful application. This makes computer science a very diverse and broad discipline. But in spite of this diversity, there are some common underlying principles. To study these basic principles, we construct abstract models of computers and computation.

Linz

Overview of Course

Loosely speaking we can think of automata, grammars, and computability as the study of what can be done by computers in principle, while complexity addresses what can be done in practice. In this book we focus almost entirely on the first of these concerns. We will study various automata, see how they are related to languages and grammars, and investigate what can and cannot be done by digital computers. Although this theory has many uses, it is inherently abstract and mathematical.

Linz

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma

A nonterminal $A \in V$ is *productive* if $A \Rightarrow^* w$ for some $w \in T^*$.

A useless production $A \rightarrow \alpha$ is one with some unproductive nonterminal in α .

For each production $A \rightarrow X_1 X_2 \cdots X_n$, A is productive if X_i for all i is either a terminal or a productive nonterminal.

A nonterminal $A = inV$ is said to be reachable if $S \Rightarrow^* \alpha A \beta$.
 S is reachable as is any A where $N \rightarrow \alpha A \beta$ and N is reachable.

Let L be a context-free language that does not contain ε . Then there exists a context-free languages can be made free of ε -productions, unit-productions, and useless productions.

1. Remove ε -productions
2. Remove unit-productions.
3. Remove useless productions.

Find all Nullable non-terminals. Replace every production with a nullable non-terminal in the RHS with two productions: one with and one without the nullable non-terminal.
If a production has n nullable non-terminals then it is replaced by 2^n productions.

Every CFG $G = (V, N, P, S)$ can be (effectively) transformed to one without cycles, non-productive or unreachable non-terminals. (This means that unit productions are unnecessary.)

- ▶ A productive non-terminal N is one for which $N \Rightarrow^* w$ for some $w \in \Sigma^*. w \in V^*$.
- ▶ A reachable non-terminal N is one for which $S \Rightarrow^* \alpha N \beta$ for some $\alpha, \beta \in (\Sigma \cup N)^*. \alpha, \beta \in (V \cup N)^*$.

All epsilon productions may also (effectively) be eliminated from a CFG, if the language does not contain the empty string. If the language contains the empty string, no epsilon productions are necessary save one: $S \rightarrow \epsilon$. $S \rightarrow \epsilon$.

Outline

Goal

History

Models of Computation

Motivating Finite Automata

Deterministic Finite Automata

Variations on Automata

Pushdown Automata

Turing Machine

Other Computational Models

Overview

Courses

Grammars

Pumping Lemma

- ▶ $A \Rightarrow B$ is the same as $(\text{not } B) \Rightarrow (\text{not } A)$
- ▶ $\text{not } \forall x A(x) \Rightarrow B(x)$ is the same as $\exists x A(x)$ and $\text{not } B(x)$
- ▶ $\text{not } \exists x A(x)$ and $B(x)$ is the same as $\forall x A(x) \Rightarrow \text{not } B(x)$

For all formal languages L , the pumping lemma holds:

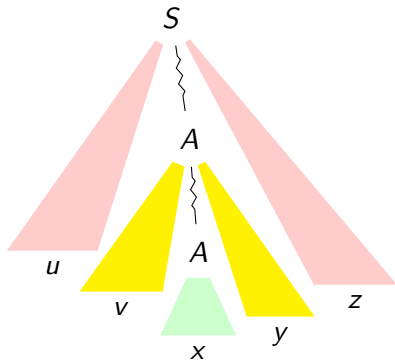
$$\text{Regular}(L) \Rightarrow \left[\begin{array}{l} \exists m \geq 1 \text{ and} \\ \forall w \in L \left(|w| > m \Rightarrow \right. \\ \quad \left. \exists x, y, z \in \Sigma^* \left[(w = xyz \text{ and } |xy| \leq m \text{ and } |y| \geq 1) \text{ and} \right. \right. \\ \quad \left. \left. \forall i \geq 0 (xy^i z \in L) \right] \right) \end{array} \right]$$

For all formal languages L , the contrapositive of the pumping lemma must hold:

$$\left[\begin{array}{l} \forall m \geq 1 \Rightarrow \\ \exists w \in L \left(|w| > m \text{ and} \right. \\ \quad \forall x, y, z \in \Sigma^* \left[(w = xyz \text{ and } |xy| \leq m \text{ and } |y| \geq 1) \Rightarrow \right. \\ \quad \quad \left. \exists i \geq 0 (xy^i z \notin L) \right] \left. \right) \\ \left. \right] \Rightarrow \text{not Regular}(L) \end{array}$$

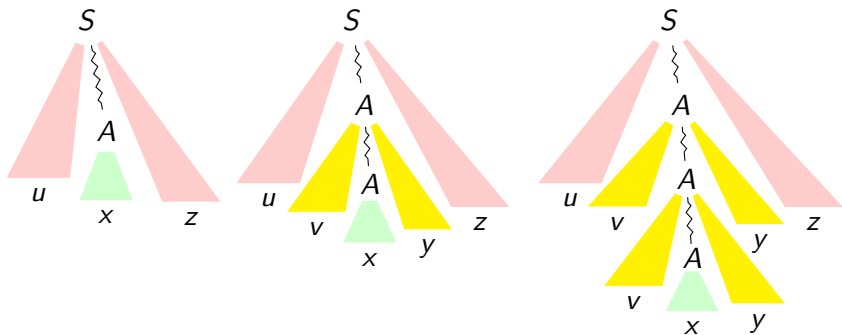
Proving a Language is Not Regular

- ▶ The adversary pick a number $m \geq 1$.
- ▶ We pick a string in L with length greater than m .
- ▶ The adversary picks strings x, y, z such that $xyz = w$, $|xy| \leq m$, and $|y| \geq 1$.
- ▶ We pick a number i such that xy^iz is not in L .
- ▶ We win, if we have a winning strategy; i.e., $xy^iz \notin L$ no matter what choices the adversary makes.



The derivation tree for the derivation:

$$S \Rightarrow^* uAz \Rightarrow^* uvAyz \Rightarrow^* uvxyz$$



The derivation tree for the derivation:

$$S \Rightarrow^* uAz \Rightarrow^* uxz$$

$$S \Rightarrow^* uAz \Rightarrow^* uvAyz \Rightarrow^* uvxyz$$

$$S \Rightarrow^* uAz \Rightarrow^* uvAyz \Rightarrow^* uvvxyyz$$

For all context-free languages L , the pumping lemma holds:

$$\text{CFL}(L) \Rightarrow \left[\begin{array}{l} \exists m \geq 1 \text{ and} \\ \forall w \in L \left(|w| > m \Rightarrow \right. \\ \quad \left. \exists u, v, x, y, z \in \Sigma^* \left[(w = uvxyz \text{ and } |vxy| \leq m \text{ and } |vy| \geq 1) \text{ and} \right. \right. \\ \quad \quad \left. \left. \forall i \geq 0 (uv^i xy^i z \in L) \right] \right) \end{array} \right]$$

For all formal languages L , the contrapositive of the pumping lemma must hold:

$$\left[\forall m \geq 1 \Rightarrow \right. \\ \left. \exists w \in L \left(|w| > m \text{ and} \right. \right. \\ \left. \left. \forall x, y, z \in \Sigma^* \left[(w = uvxyz \text{ and } |vxy| \leq m \text{ and } |vy| \geq 1) \Rightarrow \right. \right. \right. \\ \left. \left. \left. \exists i \geq 0 (uv^i xy^i z \notin L) \right] \right) \right] \Rightarrow \text{not CFL}(L)$$

Proving a Language is Not Context-Free

- ▶ The adversary pick a number $m \geq 1$.
- ▶ We pick a string in L with length greater than m .
- ▶ The adversary picks strings u, v, x, y, z such that $uvxyz = w$, $|uxy| \leq m$, and $|vy| \geq 1$.
- ▶ We pick a number i such that $uv^i xy^i z$ is not in L .
- ▶ We win, if we have a winning strategy; i.e., $uv^i xy^i z \notin L$ no matter what choices the adversary makes.

References I



Haskell Brooks Curry and Robert Feys.

Combinatory Logic.

Studies in logic and the foundations of mathematics.

North-Holland, Amsterdam, 1958.



Martin Davis, editor.

*The Undecidable: Basic Papers on Undecidable Propositions,
Unsolvability Problems, and Computable Functions.*

Dover Publication, 1965.



Martin Davis.

The Universal Computer: The Road from Leibniz to Turing.

Norton, 2000.



Maribel Fernández.

*Models of Computation - An Introduction to Computability
Theory.*

Undergraduate Topics in Computer Science. Springer, 2009.

References II



Jacques Herbrand.

Recherches sur la théorie de la démonstration.

PhD thesis, Université de Paris, Paris, France, 1930.

Translation of Chapter 5 appears in [19] pages 525–581.



James Roger Hindley and Jonathan Paul Seldin.

Introduction to Combinators and Lambda Calculus.

London Mathematical Society Student Texts #1. Cambridge University Press, 1986.







James Roger Hindley and Jonathan Paul Seldin.

Introduction to Combinators and Lambda Calculus.

Cambridge University Press, Cambridge, England, second edition, 2008.

References III

-  John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman.
Introduction to Automata Theory, Languages, and Computation, Second Edition.
Addison-Wesley, 2001.
-  John E. Hopcroft and Jeffrey D. Ullman.
Introduction to Automata Theory, Languages and Computation.
Addison-Wesley, 1979.
-  Peter Linz.
An introduction to formal languages and automata.
Jones and Bartlett Learning, fifth edition, 2012.
-  Peter Linz.
An introduction to formal languages and automata.
Jones and Bartlett Learning, sixth edition, 2017.

References IV



Marvin Lee Minsky.

Computation: Finite and Infinite Machines.

Prentice Hall, New York, 1967.



Emil Leon Post.

Formal reductions of the general combinatorial decision problem.

American Journal Mathematics, 65(2):197–215, April 1943.



Arto Salomaa.

Computation and Automata, volume 25 of *Encyclopedia of Mathematics and Its Applications*.

Cambridge University Press, Cambridge, England, 1985.



John E. Savage.

Models of computation: Exploring the power of computing.

Addison-Wesley, Reading, Massachusetts, 1998.

References V



Moses Schönfinkel.

On the building blocks of mathematical logic.

In Jean van Heijenoort, editor, *From Frege to Gödel*,
Cambridge, Massachusetts, 1977. Harvard University Press.



Ryan Stansifer.

The Study of Programming Languages.

Prentice-Hall, Englewood Cliffs, New Jersey, 1995.



Alan Mathison Turing.

On computable numbers, with an application to the
entscheidungsproblem.

Proceedings of the London Mathematical Society,
42(2):230–265, 1936.

A correction appeared in volume 43, pages 544–546, 1937.

References VI



Jan van Heijenoort.

From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931.

Harvard University Press, Cambridge, Massachusetts, 1967.